Robust concurreny

Implementing basic concurrency patterns.

GOLAB 2019, 2019–10–22, 14:30-16:00, Florence

Martin Czygan

About me

SWE @ubleipzig.

- Leipzig University Library is involved in a variety of open source projects in the library domain: catalogs, repositories, digitization and image interop frameworks (IIIF), data acquisition, processing and indexing tools
- Go for tools and services
- Co-organizer of Leipzig Gophers

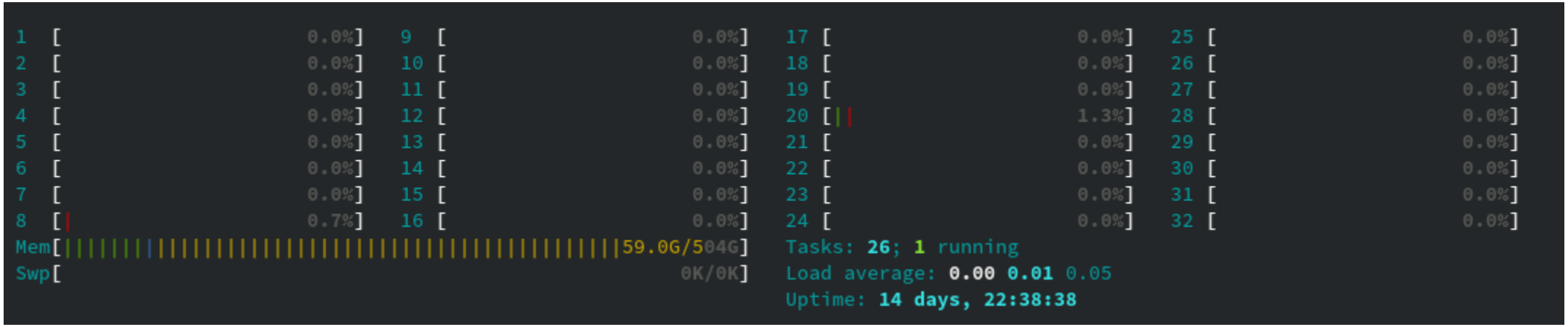
Background

Moderate use of classical concurrency tools, like threads (in Java) or multiprocessing. Use case: implementing data and legacy system access tools. Code is slow (for many reasons). Investigating parallelization approaches. Discover ZeroMQ, which looks like an embeddable networking library but acts like a concurrency framework.

Patterns, resulted in faster tools. Discovered sequential, than concurrent Go. Better resource utilization. Small parallel tools.

Background

One day.



Goal

Get more familiar with a few primitives and patterns.

- Slides
- Example code
- A few exercises
- Pop quizzes with questions related to a paper

Paper: Understanding Real-World Concurrency Bugs in Go

- Looked at Docker, k8s, etc, CockroachDB, gRPC-Go, BoltDB

> We made nine high-level key observations of Go concurrency bug causes, fixes, and detection.

Axes:

- Classic vs CSP style primitives
- Blocking vs Nonblocking bugs

If not noted otherwise, "Paper" will refer to this paper.

A few little projects

- A parallel link checker.
- A generic parallel line processing function.
- Fan-out indexing with solrbulk and esbulk.
- Hedged request with networked version of the fortune command.

Concurrency is hard

It's so hard, you may want to avoid it completely.

- Example: x/hard

Advice from https://golang.org/ref/mem

> Don't be clever.

Example: x/hard

Question: What do you think can happen?

Example: x/hard

- nothing is printed
- value = 0
- value = 1

Example: x/hard

> Most of the time, **data races are introduced because the developers are thinking about the problem sequentially**. They assume that because a line of code
> falls before another that it will run first. They assume the goroutine above will be scheduled and execute before the data variable is read in the if statement. (CIG)

Go memory model

> Many compilers (at compile time) and CPU processors (at run time) often make some optimizations by **adjusting the instruction orders**, so that the instruction execution orders **may differ from the orders presented in code**. **Instruction ordering** is also often called **memory ordering** (GO101)

Go memory model

Happens-before.

> Within a single goroutine, reads and writes must behave as if they executed in the order specified by the program. That is, compilers and processors may reorder the reads and writes executed within a single goroutine only when the reordering does not change the behavior within that goroutine as defined by the language specification.

Within a single goroutine, the happens-before order is the order expressed by the program.

Go memory model

> Within a single goroutine, there is no concurrency, so the two definitions are equivalent: a read r observes the value written by the most recent write w to v. When multiple goroutines access a shared variable v, they **must use synchronization events to establish happens-before conditions that ensure reads observe the desired writes**.

Example: x/hardsleep

```
        // Note: Extensive testing found 5µs to be the ideal time to have chance to
        // observe different results on subsequent runs. Works on my machine.
        time.Sleep(5 * time.Microsecond)
```

Example: x/hardsleep

> The takeaway here is that you should always target logical correctness. Introducing sleeps into your code can be a handy way to debug concurrent programs, but they are not a solution. (KCB)

Pop quiz

The paper looks at goroutine creation sites (go keyword). Named and anonymous functions can be used.

Question: What is more frequent?

- A: named functions
- B: anonymous functions
- C: depends on the project

Pop quiz

Question: What is more frequent?

- B: anonymous functions
- C: depends on the project

Only BoltDB is different, otherwise anonymous functions seem to be more popular.

Go concurrency primitives

Go support classic and CSP style. Which one do you choose? (KCB)

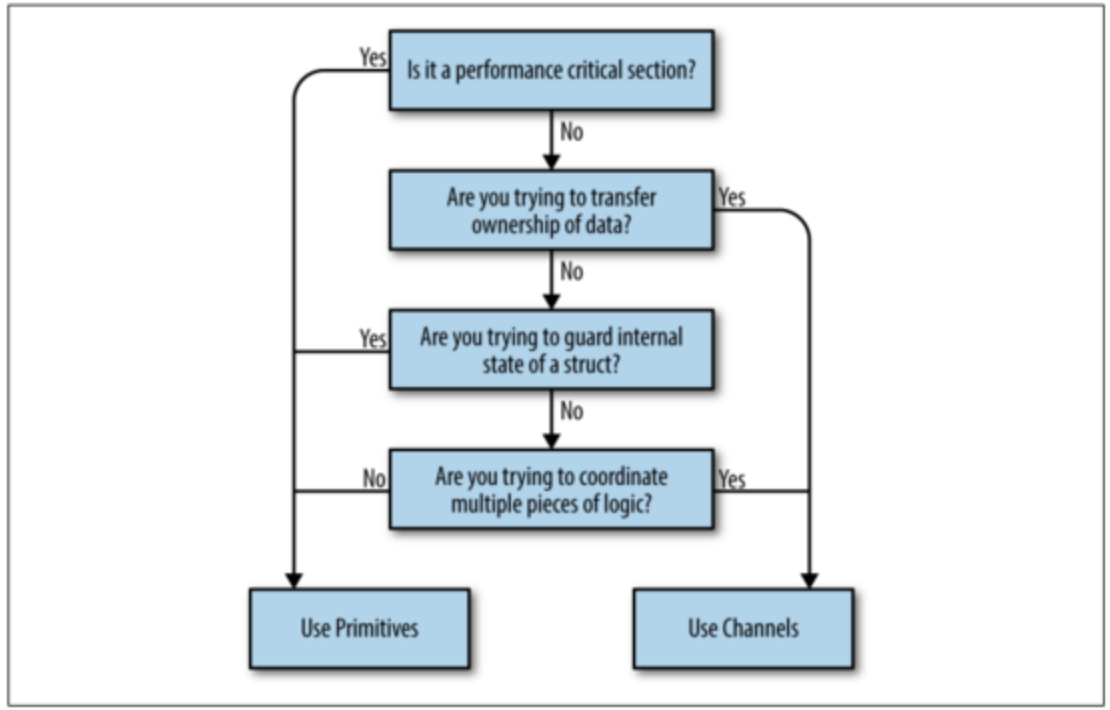


Figure 2-1. Decision tree

Pop quiz

The gRPC project has several implementations, e.g. C or Go. The Paper discribes the ratio of **normal source lines** and **goroutine / thread creation sites** as goroutine / thread creations per 1000 lines of code. Example: 1 go statement per 1000 lines would be: 1.

Question 1: What do you think this ratio is for the gRPC-Go project?

- A: 0.083
- B: 0.83
- C: 8.3

Question 2: Is the ratio for gRPC-C higher or lower?

- A: higher
- B: lower

Pop quiz

Answer: B, B.

- gRPC-Go: 0.83, gRPC-C: 0.03

It seems that people use the facilities available.

Classic Data Race

Informal definition: A data race occurs, if multiple threads access a shared resource (variable) and at least one of the accesses is write.

Exercise

Edit: x/counter/racy.go

```
// Exercise: Update a variable from different goroutines.
//
// (1) Start 100 goroutines.
// (2) Each goroutine should increment the counter variable (c).
// (3) Before main exits, print the value of c.

func main() {
        var c int // Count this up from different goroutines.
}
```

Exercise

The printed result will be inconsistent (e.g. 93, 96, 100, ...).

The race detector

The Go has a builtin race detector. It can detect these kind of errors.

- https://golang.org/doc/articles/race_detector.html

> Data races are among the most common and hardest to debug types of bugs in concurrent systems. A data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write.

Example

```
[x/counter] $ go run -race racy.go
$ go run -race racy0.go
WARNING: DATA RACE
Read at 0x00c0000b4010 by goroutine 8:
  main.main.func1()
      /home/tir/code/miku/concgo/solutions/counter/racy0.go:12 +0x38

Previous write at 0x00c0000b4010 by goroutine 7:
  main.main.func1()
      /home/tir/code/miku/concgo/solutions/counter/racy0.go:12 +0x4e

Goroutine 8 (running) created at:
  main.main()
      /home/tir/code/miku/concgo/solutions/counter/racy0.go:11 +0x83

Goroutine 7 (finished) created at:
  main.main()
      /home/tir/code/miku/concgo/solutions/counter/racy0.go:11 +0x83
```

Race detector

The race detector uses another library called thread sanitizer (TSan) and works by inserting extra statements for tracking into the executable. The executable gets slower and will consume more memory.

Race detector limits

> The **race detector only finds races that happen at runtime**, so it can't find races in code paths that are not executed. If your tests have incomplete coverage, you may find more races by running a binary built with -race under a realistic workload.

But it found numerous bugs in the standard library and elsewhere. The `-race` flag can be used during testing, in production the overhead might be considerable.

Fixing the counter

Use a `sync.Mutex`, which serializes access to a shared variable, it has two methods:

- Lock
- Unlock

> We can define a block of code to be executed in mutual exclusion by surrounding it with a call to Lock and Unlock as shown on the Inc method.

We can also use `defer` to ensure the mutex.

Exercise: x/counter

Edit: x/counter/save.go

```
// Exercise: Write a save counter.
//
// (1) Create a type counter that wraps an int and uses a sync.Mutex to protect access to the value.
// (2) Create a method on the type named `Inc` that increment the counter by one.
// (3) In main, create a counter, and start 100 goroutines, each incrementing the counter.
// (4) Print out the final value of the counter (by just accessing the struct field).
//
```

Counter wrap up

- sync.Mutex for protecting access to a resource
- A mutex will have performance implications, but these may matter later
- A sync.RWMutex allows multiple reads, but only a single write
- You can embed a struct to make the API a bit simpler
- For counters, there are alternatives, like `atomic.AddUint64`

Excursion: Executing code only once

The paper mentions (6.1.2 Errors during Message Passing) a bug (e.g. Docker#24007)that was caused by too many `close` operations of a channel. A misuse of channel.

- Example: x/extraclosefail

What happens there?

Excursion: sync.Once

The sync.Once allows to setup things once - or tear them down once. The example is not great, but is is a possibililty.

Other uses of sync.Once

Setting up a service.

> I like to use `sync.Once` to do setup inside my handler functions when I'm writing websites and services in Go. Particularly for compiling templates, loading fairly static data from disk or a database, or even loading the latest information from a remote web service. (S78)

Excursion: sync.WaitGroup

Allows goroutines to join.

> A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished.

Excursion: sync.ErrGroup

> Package errgroup provides synchronization, error propagation, and Context cancelation for groups of goroutines working on subtasks of a common task.

- Example: x/errgroup

Bounded parallelism

Go routines are lightweight, and we can start many of them fast.

- Example: x/startgr

They also do not consume a lot of memory, around 2000 bytes.

- Example: x/grsize

Bounded parallelism

Bounded parallelism restricts the number of goroutines running at any one point. We can use
a semaphores.

> Semaphores are a very general synchronization mechanism that can be used to implement mutexes, limit access to multiple resources, solve the readers-writers problem, etc. (GOP)

There is no semaphore in the sync package, but we can emulate one with a buffered channel.

Buffered channel semaphore

- the capacity of the buffered channel is the number of resources we wish to synchronize
- the length** (number of elements currently stored) of the channel is the number of resources current being used
- the capacity minus the length of the channel is the number of free resources (the integer value of traditional semaphores)

Unbounded version

- Example: x/unbounded

Exercise: Create a bounded version

- x/bounded

```
// Exercise: Create a version which limit the number of goroutines running at
// any given time. Let's use a limit of 5.
//
// (1) Use a channel as a semaphore.
```

Worker pool

Another example for a boundednessis to use a fan out pattern. The basic idea:

- A fixed number of goroutines - workers, they will consume tasks from a queue
- The main goroutine put items on the queue
- Workers work on items as the are available, round robin

Exercise: Worker pool

- Exercise: x/workerpool

```
// Exercise: A worker pool example. Some data and a basic worker is already
// there.
//
// (1) Complete the main function, setup a number of workers, the queue.
// (2) Iterate over the data and put the strings of the queue.
```

Fan-In

We used fan out, to distribute tasks to the workers. The workers might be independent, no fan in required. As an example, in a parallel indexing tool, we do not collect any results (except for an error, maybe).

Exercise: Fan-In

Extend the implementation of the worker pool example into a full fan-out, fan-in

example: x/fanin

```
// Exercise: Worker pool with fan-in. Instead of printing the results in the
// worker, the worker will put the result on an output channel. This output
// channel should just print the data.
//
// (1) Add a function fanIn that receives the results (strings) and print the
// results (or log them).
//
// (2) Update the worker implementation: Do no log there, but put the results on
// a new results channel.
//
// (3) Update main, add required channels. Also think about the how we can wait
// sensibly let all processes finish.
```

Error handling

As a final piece in the worker pool example, let's think about error handling. Currently, no errors occur, but that is not realistic.

There are various ways to implement error handling. You could create a separate channel for errors.

Another option is to group results and error into a single (new) type and pass this new type along the output channel. The worker is free to retry operations, but ultimately is free to pass on the error - to be handled at fan in time.

Exercise: Error handling

- Exercise: x/errhandling

```
// Exercise: Worker pool with fan-in and error handling. We want to pass errors
// along the results.
//
// (1) Create a result type, that hold result and error.
//
// (2) Adjust worker, fanIn function and main to use the new result type.
//
// (3) Adjust worker, so it sometimes (e.g. in 10% of the cases) will return an
// error.
//
// (4) In the fanIn function print out the good results and count the errors,
// print out the number of errors at the end.
```

Code review and API design

Package github.com/miku/parallel is a small helper for working with typically line delimited data.

- Idea: Hide concurrency behind a sequential API.

Generators and Confinement

Save operations: immutable data, data protection and confinement.

- Examples: x/confinementadhoc, x/confinementlex

Select and nil channels

Can the nil channel be useful? Example merging two streams of values.

- Example: x/merge, zeros, hang, fix

Goroutine leaks

Goroutines cannot be killed. Other options:

- complete task
- unrecoverable error
- request to quit

Goroutine leaks

- Examples: x/leaks, leaky and quit

Hedged requests

An interesting pattern to shield against failures are hedged requests. The same request is sent to different servers, in order to mitigate potentially slow or even missing responses.

Demo: `metha-fortune`. This tool almost certainly encounters failing or slow endpoints, yet still works.

This pattern has been described in various talks, blog posts and articles.

Hedged requests

- Example: x/hedged

We want quick news, and do not necessary care about the source. Make N HTTP requests to news sources and display the first one, cancel the others.

```
 ________________________________________
/ Why Botswanas election could be       \
| decided by elephants and diamonds [BBC |
\ News - World]                          /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Using context

Added to the standard library in Go 1.7.

> Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.

> Incoming requests to a server should create a Context, and outgoing calls to servers should accept a Context. The chain of function calls between them must propagate the Context, optionally replacing it with a derived Context created using WithCancel, WithDeadline, WithTimeout, or WithValue. When a Context is canceled, all Contexts derived from it are also canceled.

Exercise: Add with context

- Exercise: x/ctxadd

```
// Exercise: Write a function addContext, that takes a context and two integers
// and returns the result and an error. Make it so the add is articially
// delayed.
//
// (1) Write a function addContext, with a context and two integers, a, b as
// parameters. It should return the sum a + b and an error. In the function
// check for cancellation via Done, return early.
//
// (2) In main(), create a new context with a timeout, call the add function and
// display a result or an error.
```

Context allows to pass values with it

> Use context Values only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions.

From the bugs paper

> Go provides many new libraries, some of which use objects that are implicitly shared by multiple goroutines. If they are not used correctly, data race may hap- pen. For example, the context object type is designed to be accessed by multiple goroutines that are attached to the context. etcd#7816 is a data-race bug caused by multiple goroutines accessing the string field of a context object.

Context in client request

Allows to set a timeout on a client request.

- Example: x/ctxclient

> Failing to call the CancelFunc leaks the child and its children until the parent is canceled or the timer fires.

Context in request

Allows to cancel computation, when client disconnects.

- Example: x/ctxserve

Wrap up

We saw a few patterns for combining concurrency primitives. As the paper on real world go bugs suggests, concurrency does not get simpler to implement with CSP. But is certainly allows to approach problems differently, and often more intuitively.

References

- CIG: Concurrency in Go, Katherine Cox-Buday (2017)
- The Go documents
- GO101: https://go101.org/ and https://go101.org/article/memory-model.html
- S78: https://medium.com/statuscode/how-i-write-go-http-services-after-seven-years-37c208122831
- GOP: http://www.golangpatterns.info/concurrency/semaphores